20 | 关联查询：如何提升多表Join能力?

2020-09-23 王磊

分布式数据库30讲 进入课程 >

讲述：王磊
时长 19:24 大小 17.77M

你好，我是王磊，你也可以叫我 Ivan。

今天，我们会继续学习查询场景中的处理技术。这一讲的关键词是“多表关联”，也就是数据库中常见的 Join 操作。无论是单体数据库还是分布式数据库，关联操作的语义始终没有变，一些经典算法也保持了很好的延续性。

关联算法作为一个稍微细节点的设计，在不同数据库中是有差异的，我们还是秉承课程的整体思路，不陷入具体的设置参数、指令等内容。这样安排的依据是，只要你掌握关联算法的基本原理，就能快速掌握具体数据库的实现了。同时，有了这些原理作为基础，你能更容易地掌握分布式数据库的优化思路。

那么，我们先来看看这些经典的关联算法吧。

三类关联算法

常见的关联算法有三大类，分别是嵌套循环（Nested Loop Join）、排序归并（Sort-Merge Join）和哈希（Hash Join）。

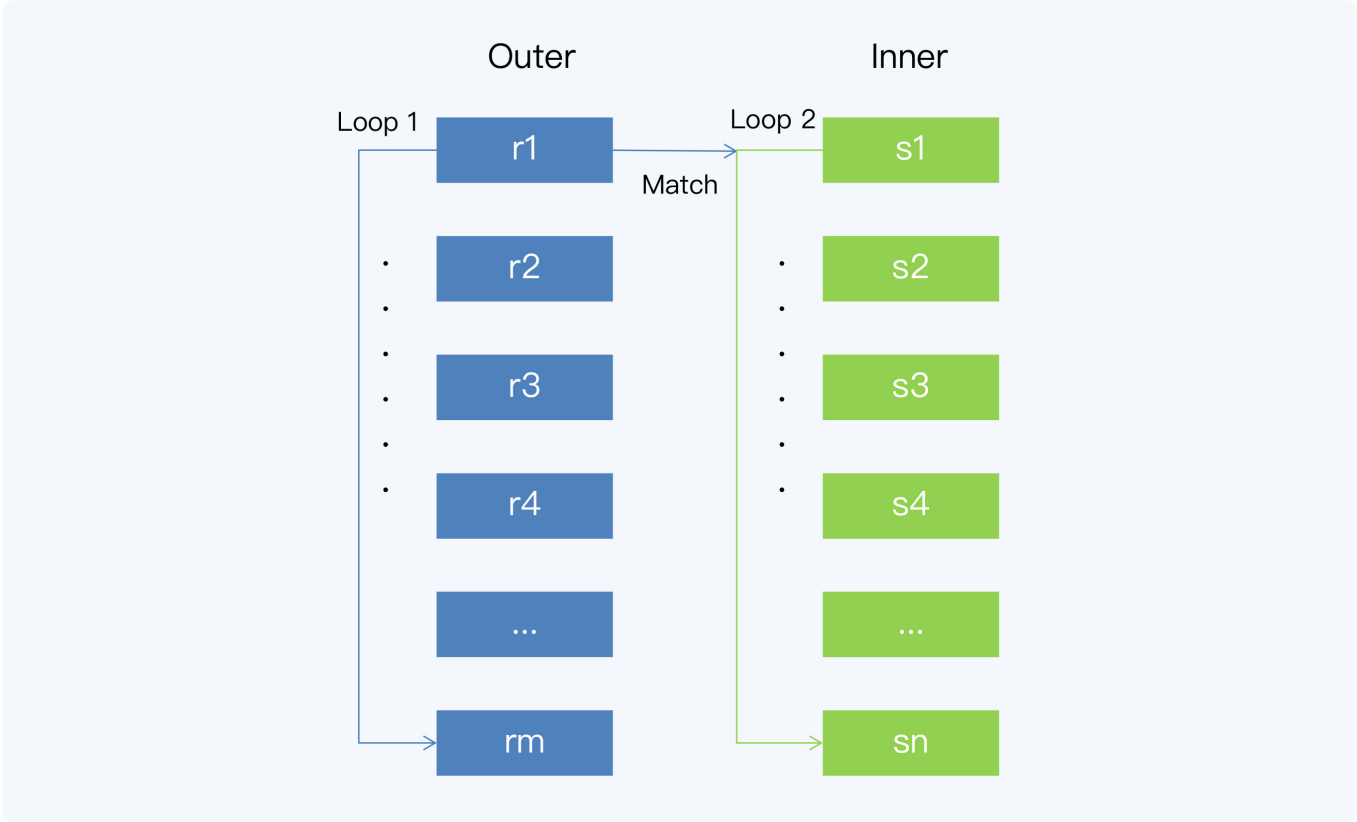
嵌套循环连接算法

所有的嵌套循环算法都由内外两个循环构成，分别从两张表中顺序取数据。其中，外层循环表称为外表（Outer 表），内层循环表则称为内表（Inner 表）。因为这个算法的过程是由遍历 Outer 表开始，所以 Outer 表也称为驱动表。在最终得到的结果集中，记录的排列顺序与 Outer 表的记录顺序是一致的。

根据在处理环节上的不同，嵌套循环算法又可以细分为三种，分别是 Simple Nested-Loop Join（SNLJ）、Block Nested-Loop Join（BNJ）和 Index Lookup Join（ILJ）。

Simple Nested Loop Join

SNLJ 是最简单粗暴的算法，所以也称为 Simple Nested-Loop Join。有些资料中会用 NLJ 指代 SNLJ。



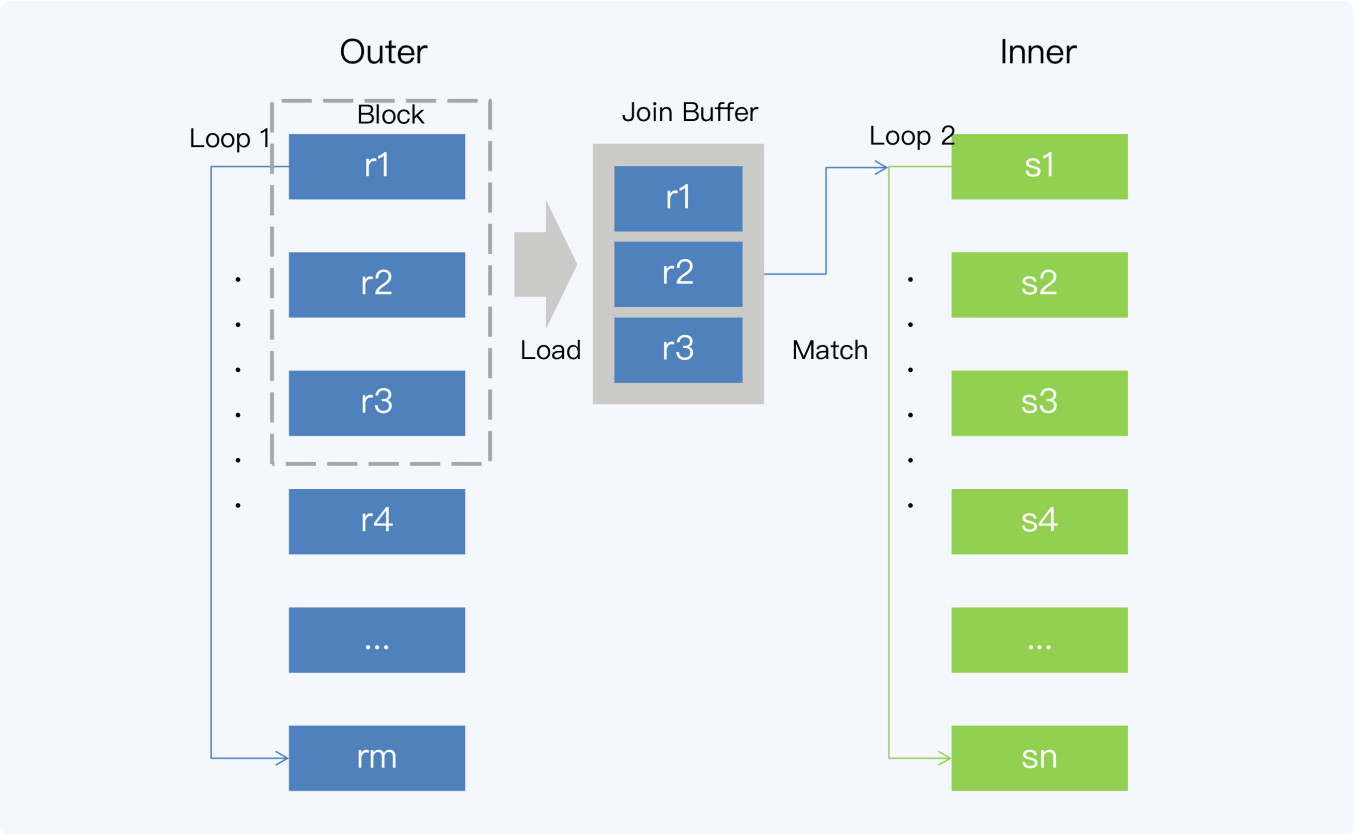
SNLJ 的执行过程是这样的：

1. 遍历 Outer 表，取一条记录 r1；
2. 遍历 Inner 表，对于 Inner 表中的每条记录，与 r1 做 join 操作并输出结果；
3. 重复步骤 1 和 2，直至遍历完 Outer 表中的所有数据，就得到了最后的结果集。

这样看，SNLJ 算法虽然简单，但也很笨拙，存在非常明显的性能问题。原因在于，每次为了匹配 Outer 表的一条记录，都要对 Inner 表做一次全表扫描操作。而全表扫描的磁盘 I/O 开销很大，所以 SNLJ 的成本很高。

Block Nested-Loop Join

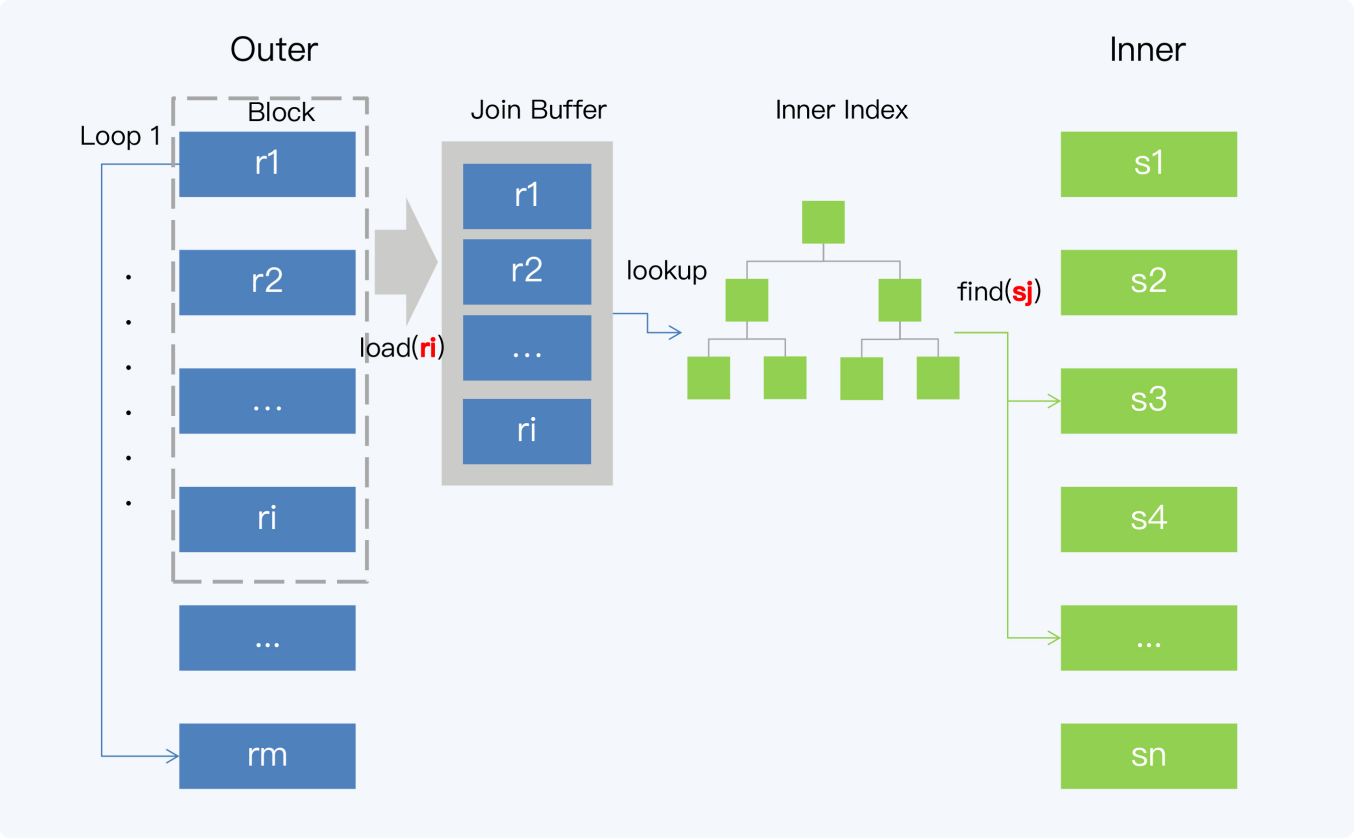
BNJ 是对 SNLJ 的一种优化，改进点就是减少 Inner 表的全表扫描次数。BNJ 的变化主要在于步骤 1，读取 Outer 表时不再只取一条记录，而是读取一个批次的 x 条记录，加载到内存中。这样执行一次 Inner 表的全表扫描就可以比较 x 条记录。在 MySQL 中，这个 x 对应一个叫做 Join Buffer 的设置项，它直接影响了 BNJ 的执行效率。



与 SNLJ 相比，BNJ 虽然在时间复杂度都是 O(m*n)（m 和 n 分别是 Outer 表和 Inner 表的记录行数），但磁盘 I/O 的开销却明显降低了，所以效果优于 SNLJ。

Index Lookup Join

你应该也注意到了，SNLJ 和 BNJ 都是直接在数据行上扫描，并没有使用索引。所以，这两种算法的磁盘 I/O 开销还是比较大的。



Index Lookup Join（ILJ）就是在 BNJ 的基础上使用了索引，算法执行过程是这样的：

1. 遍历 Outer 表，取一个批次的记录 ri；
2. 通过连接键（Join Key）和 ri 可以确定对 Inner 表索引的扫描范围，再通过索引得到对应的若干条数据记录，记为 sj；
3. 将 ri 的每一条记录与 sj 的每一条记录做 Join 操作并输出结果；
4. 重复前三步，直到遍历完 Outer 表中的所有数据，就得到了最后结果集。

看到这里，ILJ 的主要优化点也很明显了，就是对 Inner 表进行索引扫描。那么，你可能会问了为什么不让 Outer 表也做索引扫描呢?

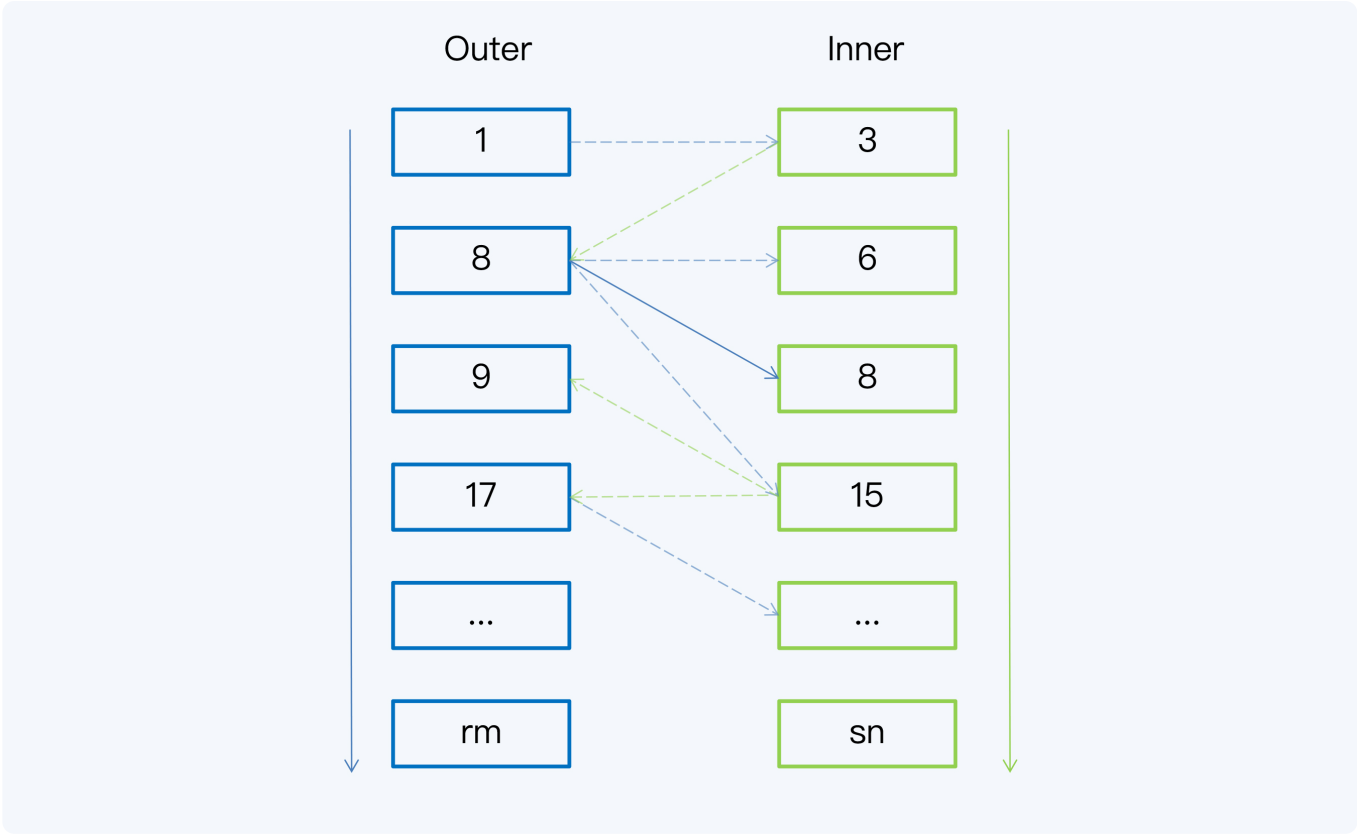
我认为，Outer 表当然也可以走索引。但是，BNJ 在 Inner 表上要做多次全表扫描成本最高，所以 Inner 表上使用索引的效果最显著，也就成为了算法的重点。而对 Outer 表来说，因为扫描结果集要放入内存中暂存，这意味着它的记录数是比较有限的，索引带来的效果也就没有 Inner 表那么显著，所以在定义中没有强调这部分。

关联算法的定义是为了让我们专注其中的重点，而不是僵化地去理解它。比如，我们会在有些教材上看到，对 ILJ 的定义就是直接在 SNLJ 增加索引，并不是在 BNJ 上拓展。而要真正在工程中应用关联算法，都要结合具体场景进一步优化。

排序归并连接算法

排序归并算法就是 Sort-Merge Join（SMJ），也被称为 Merge Join。SMJ 可以分为排序和归并两个阶段：

1. 第一阶段是排序，就是对 Outer 表和 Inner 表进行排序，排序的依据就是每条记录在连接键上的数值。
2. 第二阶段就是归并，因为两张表已经按照同样的顺序排列，所以 Outer 表和 Inner 表各一次循环遍历就能完成比对工作了。



简单来说，SMJ 就是先要把两个数据集合变成两个数据序列，也就是有序的数据单元，然后再做循环比对。这样算下来，它的计算成本是两次排序再加两次循环。你可能会觉得奇怪，这成本是不是比 NLJ 还要高呀？

是的。所以选择 SMJ 是有前提的，而这个前提就是表的记录本身就是有序的，否则就不划算了。我们知道，索引是天然有序的，如果表的连接键刚好是索引列，那么 SMJ 就是三种

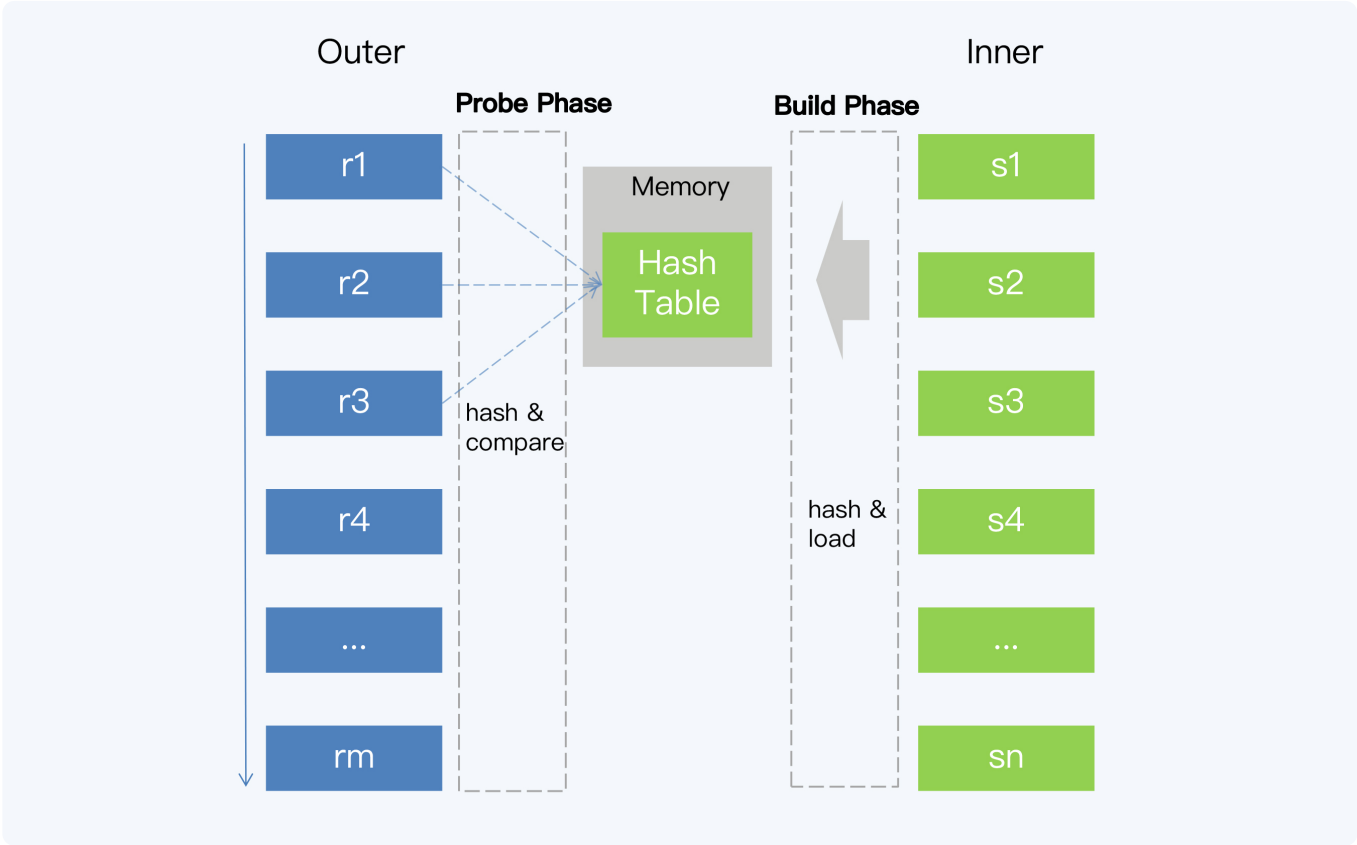
嵌套循环算法中成本最低的，它的时间复杂度只有 O(m+n)。

哈希连接算法

哈希连接的基本思想是取关联表的记录，计算连接键上数据项的哈希值，再根据哈希值映射为若干组，然后分组进行匹配。这个算法体现了一种分治思想。具体来说，常见的哈希连接算法有三种，分别是 Simple Hash Join、Grace Hash Join 和 Hybrid Hash Join。

Simple Hash Join

Simple Hash Join，也称为经典哈希连接（Classic Hash Join），它的执行过程包括建立阶段（Build Phase）和探测阶段（Probe Phase）。



1. 建立阶段

选择一张表作为 Inner 表，对其中每条记录上的连接属性（Join Attribute）使用哈希函数得到哈希值，从而建立一个哈希表。在计算逻辑允许的情况下，建立阶段选择数据量较小的表作为 Inner 表，以减少生成哈希表的时间和空间开销。

2. 探测阶段

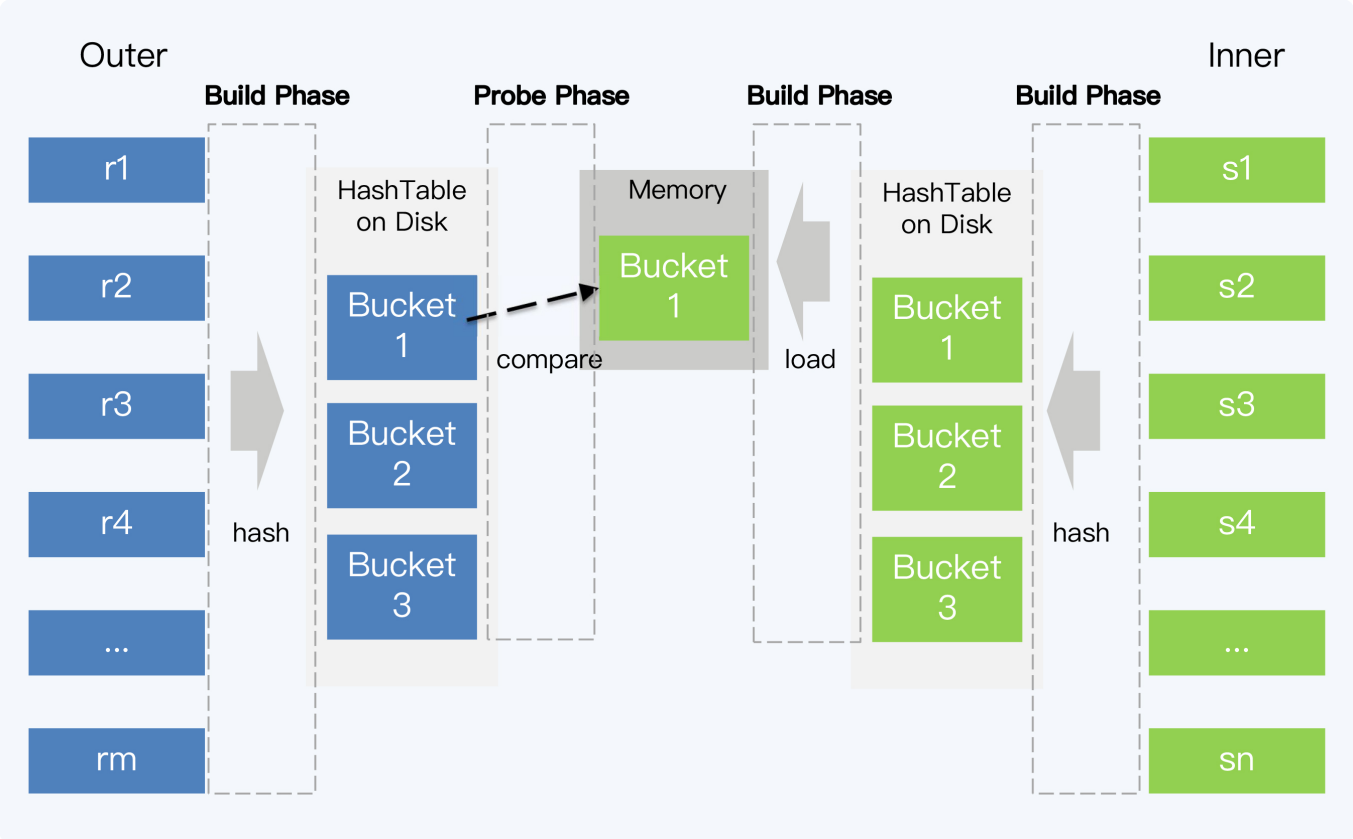
另一个表作为 Outer 表，扫描它的每一行并计算连接属性的哈希值，与建立阶段生成的哈希表进行对比。当然，哈希值相等不代表连接属性相等，还要再做一次判断，返回最终满足条件的记录。

通过 Simple Hash Join 这个命名，我们就能知道它也是一个简单的算法。这里的简单是说，它做了非常理想化的假设，也就是 Inner 表形成的哈希表小到能够放入内存中。可实际上，即使对于单体数据库来说，这个哈希表也是有可能超过内存容量的。

哈希表无法全部放入内存怎么办呢？这时就要使用 Grace HashJoin 算法了。

Grace Hash Join

GHJ 算法与 SHJ 的不同之处在于，GHJ 正视了哈希表大于内存这个问题，将哈希表分块缓存在磁盘上。GHJ 中的 Grace 并不是指某项技术，而是首个采用该算法的数据库的名字。



GHJ 算法的执行过程，也是分为两个阶段。

第一阶段，Inner 表的记录会根据哈希值分成若干个块（Bucket）写入磁盘，而且每个 Bucket 必须小于内存容量。Outer 表也按照同样的方法被分为若干 Bucket 写入磁盘，但它的大小并不受到内存容量限制。

第二阶段和 SHJ 类似，先将 Inner 表的 Bucket 加载到内存，再读取 Outer 表对应 Bucket 的记录进行匹配，所有 Inner 表和 Outer 表的 Bucket 都读取完毕后，就得到了最终的结果集。

Hybrid Hash Join

Hybrid Hash Join，也就是混合哈希，字面上是指 Simple Hash Join 和 Grace Hash Join 的混合。实际上，它主要是针对 Grace Hash Join 的优化，在内存够用的情况下，可以将 Inner 表的第一个 Bucket 和 Outer 表的第一个 Bucket 都保留在内存中，这样建立阶段一结束就可以进行匹配，节省了先写入磁盘再读取的两次 I/O 操作。

总体来说，哈希连接的核心思想和排序归并很相似，都是对内外表的记录分别只做一次循环。哈希连接算法不仅能够处理大小表关联，对提升大表之间关联的效率也有明显效果，但限制条件就是适用于等值连接。

分布式数据库实现

学习了基本的关联算法后，我们聚焦到分布式数据库的范畴内继续讨论。其实，在学习 GHJ 的过程中，你是不是已经嗅到一点分布式架构的味道了？GHJ 就是将一个大任务拆解成若干子任务并执行的过程，这些子任务本身是独立的，如果调度到不同的节点上运行，那这就是一个并行框架。由此，我们可以说，分布式架构下关联算法的优化和并行框架密切相关。

并行框架

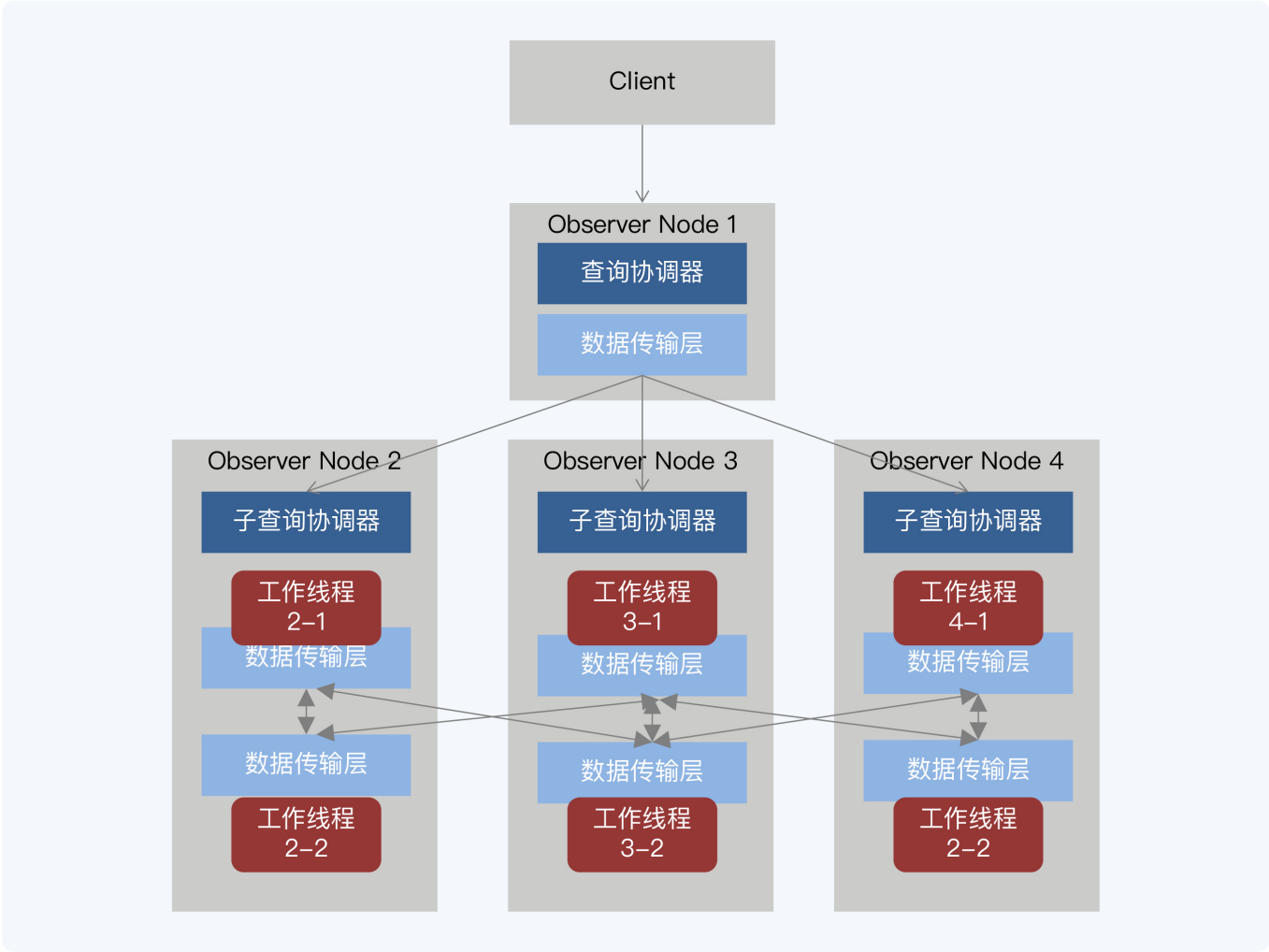
在第 19 讲中我们提到了计算下推，换个角度看，其实它就是一种并行框架，不过是最简单的并行框架。因为在很多情况下，计算任务的执行节点和对应数据的存储节点并不是完全对应的，也就没办法只依据数据分布就拆分出子任务。

那么，要想在数据交错分布的情况下，合理地划分和调度子任务就需要引入更复杂的计算引擎。这种并行执行引擎在 OLAP 数据库中比较常见，通常称为 MPP（Massively Parallel Processing）。很明显，MPP 已经超出了 OLTP 计算引擎的范畴，并不是所有分布式数据库都支持的。

比如，我们前面介绍过的 TiDB，在最初的 TiDB + TiKV 的体系中，就没有 MPP 引擎。TiDB 的存储节点之间是不能通讯的（除了 Raft 协议），这就意味着如果子任务之间有数

据传输就必须以计算节点为通道。这样，计算节点很容易成为瓶颈，同时增加了网络传输负载。由此可见，必须经过计算节点这个约束，是生成高效并行计划的一个障碍。后来，TiDB 也没有打破这个约束，而是通过引入 Spark 来处理复杂的 OLAP 计算任务，这就是 TiSpark 组件。

但并不是所有分布式数据库都采用引入外部组件的方式，比如 OceanBase 就在原有设计中拓展了并行执行框架，实现了更复杂的任务调度，在存储节点间也可以直接进行数据交换。



OceanBase 大致也是 P2P 架构，每个 Observer 部署了相同的服务，在运行过程中，动态的承担不同角色。图中一个 Observer 节点承担了入口处的查询协调器，其他节点作为子查询协调器，上面的工作线程是真正的任务执行者。

理解了并行框架的必要性，我们再回到多表关联这个具体场景。多表关联的复杂度，主要看参与表的数据量。其中，小表之间的关联都比较简单，所以我们接下来主要关注小表与大表关联和大表之间的关联。

大小表关联（复制表）

大小表关联时，可以把小表复制到相关存储节点，这样全局关联就被转换为一系列的本地关联，再汇总起来就得到了最终结果。这种算法的具体实现方式有两种。

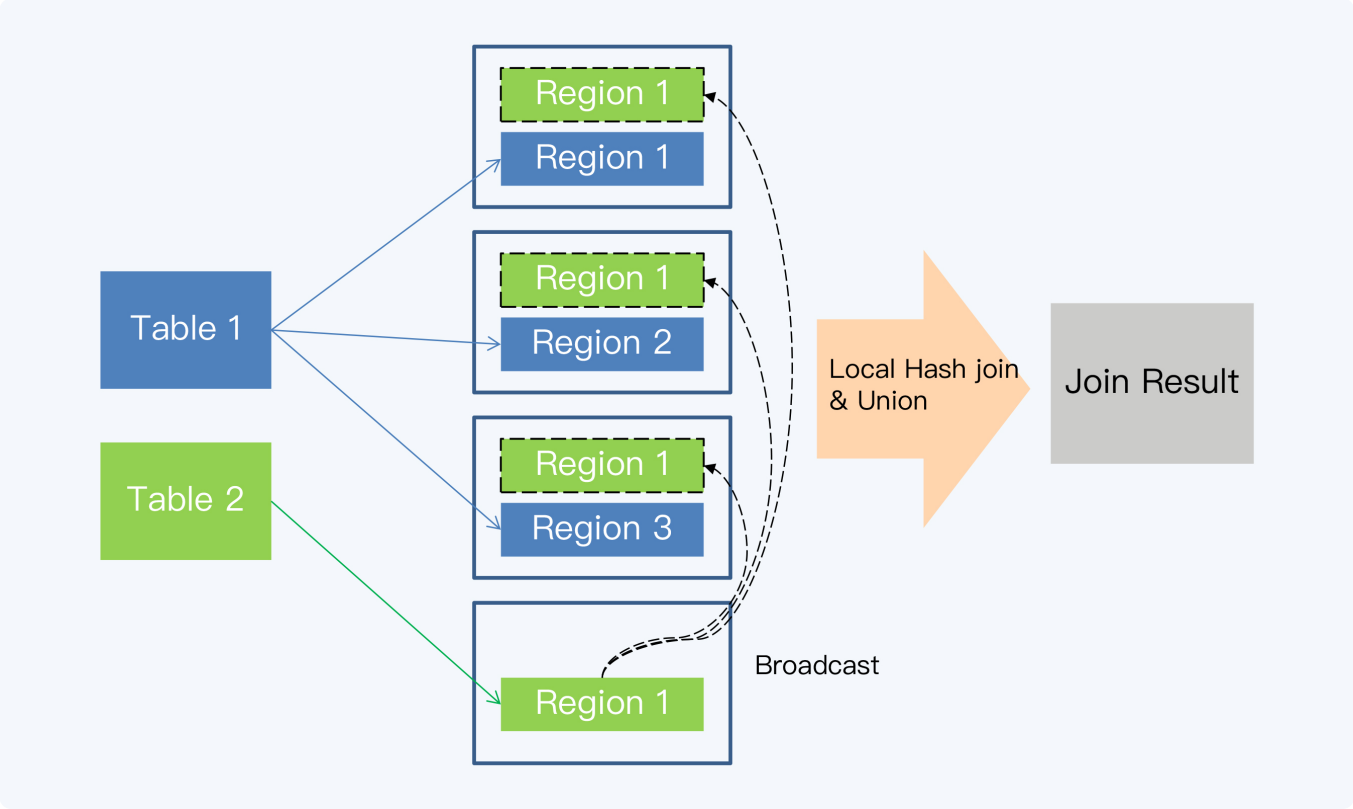
1. 静态的方式

静态的方式，其实就是在创建表的时候，直接使用关键字将表声明为复制表，这样每个节点上都会保留一份数据副本。当它与大表关联时，计算节点就可以将关联操作下推到每个存储节点进行。很多分布式数据库，比如 TBase、TDSQL 等，都支持定义复制表。

2. 动态方式

动态方式也称为“小表广播”，这种方式不需要人工预先定义，而是在关联发生时，系统自行处理。这就是说，当关联的某张表足够小时，在整个集群中分发不会带来太大的网络开销，系统就将其即时地复制到相关的数据节点上，实现本地关联。

下面这张图体现了小表广播的过程。



动态方式和并行执行引擎有直接的联系，例如 Spark 并行执行引擎中的 Broadcast Hash Join 就是先采用动态广播方式，而后在每个节点上再执行哈希连接。

当然，这里的 “复制”和“广播”只表达了自然语义，不能作为静态还是动态的判断标准。比如，TDSQL 中的“广播表”，TBase 中的“复制表”，说的都是指静态方式。

大表关联（重分布）

复制表解决了大小表关联的问题，还剩下最棘手的大表间关联，它的解决方案通常就是重分布。

我们直接看一个例子，现在要对 A、B 两张大表进行关联，执行下面的 SQL：

```
1 select A.C1,B.C2 from A,B where A.C1=B.C1;
```

复制代码

这个 SQL 可能会引发两种不同的重分布操作。

第一种，如果 C1 是 A 表的分区键，但不是 B 表的分区键，则 B 表按照 C1 做重分布，推送到 A 的各个分片上，实现本地关联。

第二种，如果两张表的分区键都不是 C1，则两张表都要按照 C1 做重分布，而后在多个节点上再做本地关联。当然这种情况的执行代价就比较高了。

这个基于重分布的关联过程，其实和 MapReduce、Spark 等并行计算引擎的思路是一样的，基本等同于它们的 Shuffle 操作。我们可以用 Spark 的 Shuffle Hash Join 来对比学习一下。

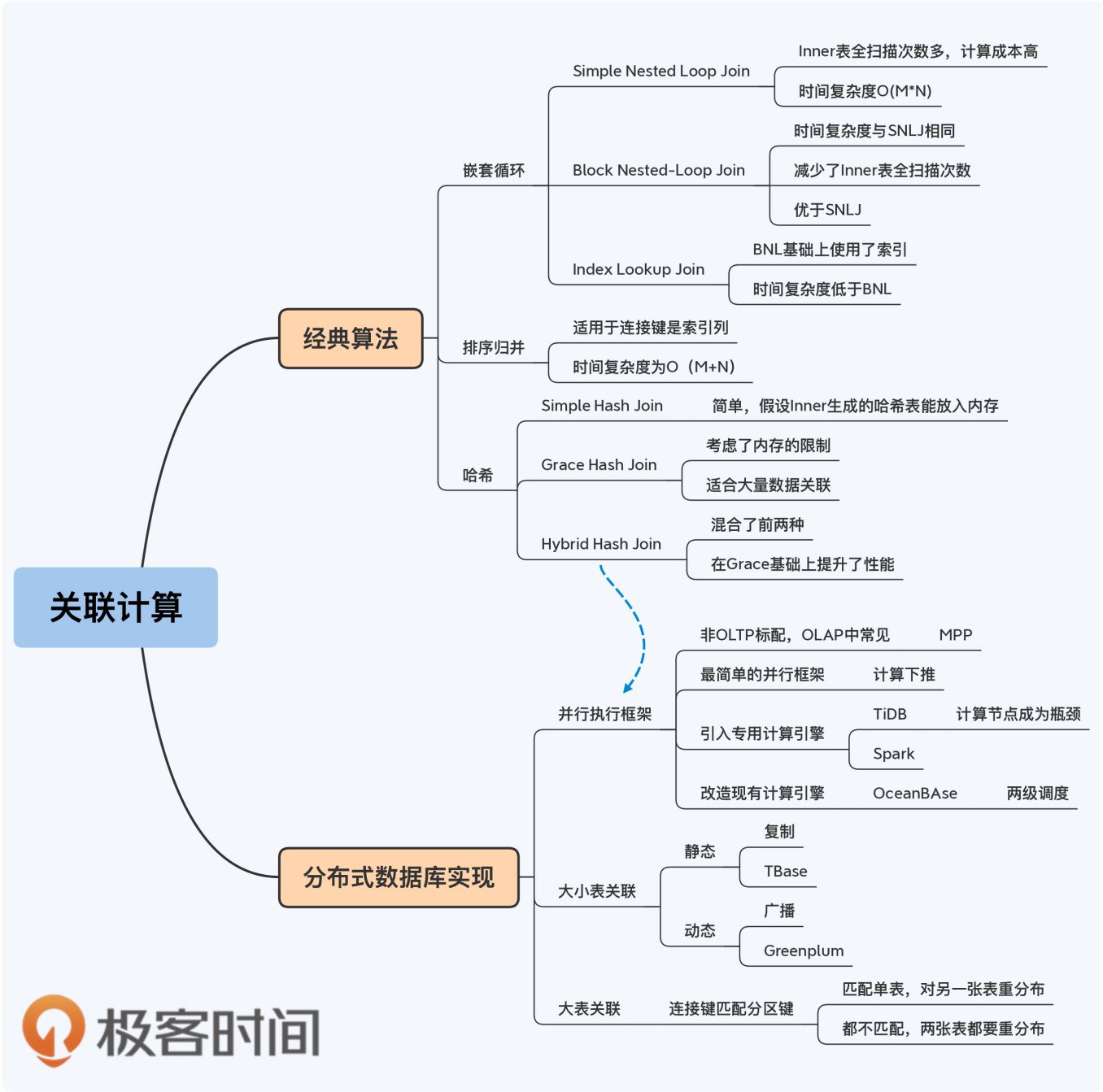
1. shuffle 阶段：分别将两个表按照连接键进行分区，将相同连接键的记录重分布到同一节点，数据就会被分配到尽量多的节点上，增大并行度。
2. hash join 阶段：每个分区节点上的数据单独执行单机 hash join 算法。

小结

那么，今天的课程就到这里了，让我们梳理一下这一讲的要点。

1. 关联是数据库中比较复杂的操作，相关算法主要分为三类，分别是嵌套循环、排序归并和哈希。嵌套循环是比较基础的排序算法，大多数数据库都会支持，又细分为 SNLJ、BNLJ、ILJ 三种。排序归并算法，仅适用于关联数据有序的情况，比如连接键是关联表的索引列时，在这个前提下排序归并算法的成本低于嵌套循环。
2. 哈希算法适用于大小表关联和大表关联的场景，并不是 OLTP 数据库的标配。在海量数据下，哈希算法比嵌套循环和排序归并这两种算法的效果更好，所以在 OLAP 数据库和大数据技术产品中比较常见。常用的哈希算法包括 SHJ、GHJ 和 HHJ。
3. 分布式数据库下关联算法的优化依赖于并行框架（MPP），而并行框架更多地出现在 OLAP 数据库中，不是分布式数据库的标配。
4. 大小表关联的方法是复制表，有静态和动态两种实现方式。静态方式是预先将小表存储在所有节点，动态方式是在关联发生时决定是否广播小表。大表间关联的方法是重分布。当 A 和 B 两张表关联时，如果 A 表的分区键与连接键相同，只需要对 B 表做单表重分布，否则两表都需要重分布，代价更大。

关联计算是查询场景中比较复杂的操作，即使面向 OLTP 场景的传统单体数据库也没有完善的处理，比如 MySQL 直到 8.0 版本才支持 Hash Join。而分布式数据库也由于自身定位不同，对关联算法支持程度存在差异。总的来说，越倾向于支持 OLAP 场景，对关联算法的支持度也就越高。



思考题

课程的最后，我们来看看今天的思考题。我在介绍哈希算法时，说“在计算逻辑允许的情况下，建立阶段选择数据量较小的表作为 Inner 表”，我的问题就是在什么情况下，系统无法根据数据量决定 Inner 表呢？

欢迎你在评论区留言和我一起讨论，我会在答疑篇和你继续讨论这个问题。如果你身边的朋友也对关联查询这个话题感兴趣，你也可以把今天这一讲分享给他，我们一起讨论。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 查询性能优化：计算与存储分离架构下有哪些优化思路?

下一篇 21 | 查询执行引擎：如何让聚合计算加速?

精选留言 (7)

写留言

Jxin
2020-09-23

重分布这个有疑惑: 如果我有多个关联查询呢?每次关联查询都重分布？这样重分布就可能是个死循环了。因为A关联查询和B关联查询的重分布,可能会相互影响。

展开

作者回复: 多表关联会被拆解成多个阶段执行，21讲正好有个例子，可以关注下。

1 1

游弋云端
2020-09-24

“在计算逻辑允许的情况下，建立阶段选择数据量较小的表作为 Inner 表”，我的问题就是在什么情况下，系统无法根据数据量决定 Inner 表呢？
答复：如果连接属性本身内容重复较多，但是表格很大，这样反而选择较大的表格作为 Inner 表，可以根据内容相同，从而节省Hash的计算资源。想到这种场景。

展开

赵见跃
2020-09-24

哈希算法，“在计算逻辑允许的情况下，建立阶段选择数据量较小的表作为 Inner 表”，我的问题就是在什么情况下，系统无法根据数据量决定 Inner 表呢？这个问题，也很困惑，大家给指点一下呀，谢谢。

有铭
2020-09-23

Grace这个最早使用GHJ算法数据库没有查到啊？

很久以前就困惑传统数据库在分表分库后如何解决joint的问题。今天看到这篇文章后豁然开朗

作者回复: Grace出处这个说法是从论文中来的，我也没有考证，应该是某个学术界的数据库吧。

piboye
2020-09-23

复制表，写的时候也是异步复制到其他节点上？

展开

licl1008
2020-09-23

数据量小的一侧不是构建侧，这种情况不能单单根据数据量来决定innner

扩散性百万咸面包
2020-09-23

我对重分布的理解：按照某个非分区键再做一遍重分区。

对于这个思考题，应该是两个关联表的大小都远远超过了单机的内存？所以必须要上分布式？

展开